Towards Multi-server Unix Emulation on Mach: Design and Implementation of a File Server Architecture

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
D. Raghuram

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
January, 1998

A 124940

CSE-1998-M-RAG-TOW

Entered in System

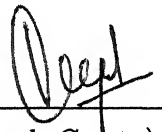
Dm
20-4-98

A124940

POST GRADUATE OFFICE
I.I.T Kanpur.
Submitted on 05.01.98

Certificate

Certified that the work contained in the thesis entitled " *Towards Multi-server Unix Emulation on Mach: Design and Implementation of a File Server Architecture*", by Mr.*D. Raghuram*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



(Dr. Deepak Gupta)
Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

January, 1998

Abstract

This work is a beginning towards our efforts to build a multi-server emulation of Unix over Mach. In this thesis, we present the design of a file server architecture for emulating the Unix file services. We have chosen to have a separate file server for each type of file system in order to improve the modularity of the system. The design which is presented is applicable to all types of file systems. Currently the rest of the services are provided by a thinned down version of the Lites server which is a single server Unix emulation on Mach. An implementation of a server for the fast file system of BSD is also presented. Experiments show that the performance of this server is comparable to that of the original Lites server.

Acknowledgments

I am greatly indebted to my supervisor Dr. Deepak Gupta for all the help he had extended in completing the thesis. I thank him whole-heartedly. I would like to thank the examiners Dr. Rajat Moona and Dr. Sanjay Mittal for making helpful suggestions regarding the thesis report. My friends made my stay at IITK a cherishable one. It will not be without a twinge of nostalgia that I will look back on these days when we had so much fun together. I would like to thank each one of them. Finally the campus, with its beautiful surroundings and natural settings, had been the reason on more than one occasion, for the feelings of joy which are beyond expression. I thank everything that formed part of this environment.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The original Unix kernel, in its initial stages, was very small. But with the addition of new components like networking, the kernel evolved into a big program and its maintainability became quite difficult [LMKQ89]. Designing the kernel in a modular way helped to an extent, to manage it and to add new services. But still its size made adding new services difficult. To facilitate easy addition, debugging, and testing of new services, Accetta et al [ABB+86] proposed that the kernel should provide minimal services and all the higher level services like files and sockets should be implemented as user-level servers on this kernel.

1.1 Microkernels

Traditionally operating systems have been designed and implemented in a monolithic fashion, as single big programs which provide all the services to user applications. Even if designed in a modular fashion to some extent, the freedom with which any function can be called, leads to complex code. Adding new services usually involves understanding a major part and further it can not be done while the system is being used. The entire kernel is usually non-pageable [Sta90].

To make matters simple, the idea of a *Microkernel* — where the kernel provides most basic abstractions while all higher level services are implemented as user level processes — gained ground. In such an architecture, the kernel provides only those

services that *have* to be implemented in the kernel. Such services include task and thread management, CPU scheduling, memory management, IPC and, device management etc. The rest of the services are implemented as user level processes. This design isolates the code for the various services thereby making the whole system much more manageable and maintainable. In such a design, to access a service, an application program has to send a message to the server using IPC facilities of the underlying kernel, instead of making a system call[1].

Further, portability is improved as all the machine dependent code is now concentrated in a small microkernel. Very little machine dependent code now exists in user level servers. Debugging and testing of servers offering new services can be carried out without the need to reboot the system.

The traditional operating system services can now be provided by running one or more user-level servers. The existing application binaries can be emulated by placing an emulation library in each application's address space. The emulator code communicates with the servers concerned and thus services the system calls. For this to work, the microkernel should provide the facility to redirect system call traps to a location in user address space. It is also possible to have multiple operating system environments by running different servers for each operating system. For example, we can have Linux, BSD4.4, MS-DOS all running simultaneously.

In the monolithic kernel architecture, application programs invoke a system call by executing special hardware instructions like *trap*, which causes control to be transferred to the kernel. In the microkernel architecture, on the other hand, the application program has to send a message to a user level server. This leads to some performance penalty, as context switching between address spaces is more expensive than trapping into the kernel. But with rapid strides being made in hardware speeds, the extra modularity is worth a little fall in performance.

Mach [ABB+86], QNX [Hil92], Chorus [Roz92], and Amoeba [vR92] are some examples for microrkernels.

[1]To use the IPC facility of the kernel, one would still have to make a system call. This call would, however, be different than the one made in the case of a monolithic kernel.

1.2 Unix as an Application Program

It is both possible and rational to think of Unix not as an operating system kernel but as an application program — a server or set of servers that can provide client programs with specific programming abstractions [G+90]. The functions of a Unix system can be cleanly divided into two parts: facilities which provide hardware support, scheduling, communication and virtual memory, and support for higher level services like files and sockets. While a microkernel such as Mach can provide functionality of the first part, the higher level abstractions can be implemented by a single server or a set of servers with a transparent emulation library placed in application's address space. Existing Unix emulation implementations over Mach include Lites [Hel94], OSF/1 , and UX [G+90]. These are all single server implementations.

1.3 Single server vs Multi-server Emulation

The user level servers providing higher abstractions can be structured in different ways. There can be just one server or multiple servers that interact. A single server design has the advantage of using the existing code base better, apart from being simple. In a multi-server design, each component is simpler and easier to understand but at the cost of increased complexity in inter-operating. It is relatively more difficult to understand the overall structure of system and debugging becomes very difficult. Nevertheless, multi-server paradigm offers most flexible design in that only those services which are required need to be run. It is easier to make changes and add functionality.

1.4 The Present Work

The presented work is a beginning of our efforts in the direction of implementing a multi-server emulation of Unix on Mach. There are two ways in which this can be done. One way is to develop all the servers from scratch (reusing code from a traditional Unix kernel or from a single-server Unix emulation, wherever possible). Due to various reasons, this approach is not possible for us. Instead, we have chosen

to start with an existing Unix emulation server (Lites) and extract code from it piecemeal.

In the described work, we have taken out the file server functionality from Lites and built an independent file server to replace this functionality. Thus, we now have a two server Unix emulation — the new file server, providing file system related services, and the reduced Lites server which provides the rest of Unix functionality. As the project progresses, code for other services will be taken out from the Lites server and put into new single functionality servers.

1.5 Related Work

Hurd is GNU's Unix like operating system which runs on Mach. It is a multi-server system where GNU C library communicates with different servers to provide Unix semantics [Sta90]. The servers include the authentication server — where each port identifies a user, the process server — which acts as an information categorization repository, an I/O server that provides the terminal semantics of POSIX, an exec server which loads binaries as part of the system call *execve*, and the file servers for different types of file systems. The Hurd is not yet a stable system. Its initial version was released only in August 1996.

1.6 Organization of the Report

The rest of the report is organized as follows. In Chapter 2, an introduction of Mach and Lites is presented. Design is covered in Chapter 3. Chapter 4 details the implementation while Chapter 5 contains performance studies. Finally, Chapter 6 contains conclusions.

Chapter 2

Mach and Lites

2.1 Mach

The *Mach microkernel* is designed to incorporate many recent innovations in operating system research to produce a fully functional, technically advanced system. Mach incorporates multiprocessing support. Mach 3.0 microkernel designed at the *Carnegie Mellon University* is a stable and mature platform and serves as a base of many current operating system research projects.

2.1.1 Design Goals and Chief Features

The key design goals and features of Mach are as follows:

- *Multiprocessor operation:* Mach was designed to execute on a shared memory multiprocessor. Mach provides a multi-threaded model of user processes, with execution environments called *tasks*. *Threads* are pre-emptively scheduled, whether they belong to the same task or to different tasks, to allow for parallel execution on a shared memory multiprocessor.

- *Transparent extension to network operation:* Mach has adopted a location-independent communication model involving *ports* as destinations. The kernel itself is network unaware and as far as it is concerned, all ports are local (i.e, all ports belong to tasks running on the same machine). However, using a user

level network server, one can transparently communicate with tasks running on other machines in the network.

- *User-level servers:* Mach supports an object-based model in which resources are managed either by the kernel or by user-level servers. With the exception of virtual address space and port name space, all other resources are accessed uniformly by message passing, irrespective of how they are managed. To every resource, there corresponds a port managed by a server. The *Mach Interface Generator*(MiG) was developed to generate RPC stubs used to hide message-based accesses at the language level [Loe91].

- *Operating system emulation:* To support the binary-emulation of Unix and other operating systems, Mach allows for transparent redirection of system calls to a given address in the task. Usually an emulation library lies at this address which sends an appropriate message to the appropriate server. These operating system servers lie on top of the microkernel as shown in Figure 1.

- *Flexible virtual memory implementation:* Mach supports a large, sparse process address space, possibly containing many regions. Both messages and open files, for example, can appear as virtual memory regions. Mach was designed to allow user-level servers to implement backing storage for virtual memory pages.

- *Portability:* Mach was designed to be portable to a variety of hardware platforms. For this reason, machine-dependent code has been isolated as far as possible. In particular, the virtual memory code has been divided between machine-independent and machine-dependent parts.

2.1.2 Basic Abstractions

The basic abstractions provided by the Mach kernel are as follows.

- **Tasks:** A Mach task is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and port name space. A task may contain one or more threads.

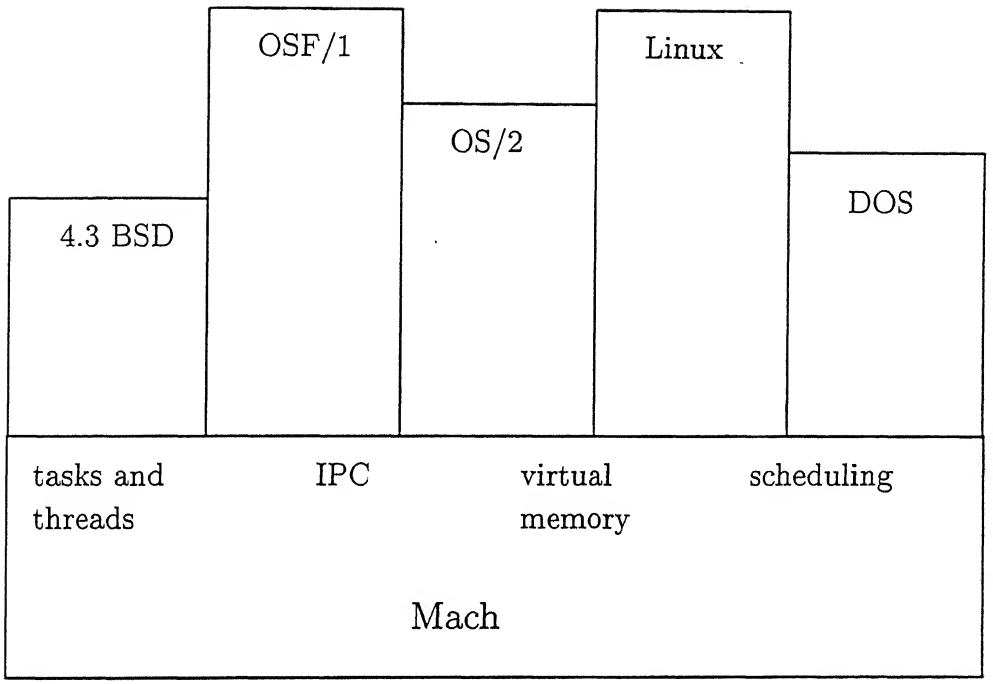


Figure 1: Mach 3 Structure

- **Threads:** A thread is a basic unit of execution, and must run in the context of a task (which provides the address space). All threads within a task share the resources (ports, memory, and so on) of the task. A traditional Unix process would be equivalent to a Mach task with a single thread of control.

- **Ports:** A port is a unidirectional communication channel and serves as a basic object reference mechanism in Mach. A port is implemented as a kernel-protected queue of messages. A send operation enqueues the message while a receive operation dequeues it. Tasks should have *port rights* — kernel managed capabilities — to send or receive messages on a port. A task specifies port rights through *port names*, which are indices into the *port name space*. There are three types of port rights.

  receive right Allows the holder to receive messages from the associated port. Only one task may hold a receive right at any given point of time.

send right Allows the holder to send messages to the associated port.

Send-once right Allows the holder to send a single message to the associated port. The right self-destructs after the message is sent.

The programmer invokes an operation on an object by *sending* a message to a port associated with the object. The object being represented by a port *receives* the messages.

- **Port sets:** A port set is a group of ports that can be treated as a single unit when receiving a message. A thread can receive messages from a port set, and thus service multiple ports. Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message.

- **Messages:** A message is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain the actual data or a pointer to out-of-line data. Port rights can be passed in messages; in general, passing port rights in messages is the only way to move them among tasks.

- **Virtual Memory:** Mach provides a task with a large, potentially sparse address space. An address space contains memory regions. Mach allows the backing storage to any memory region to be implemented by a user task known as an *External Memory Manager*. A *Memory object* is a port that backs some virtual memory pages. The physical memory acts as a cache of memory objects. Mach sends messages to memory object ports when pages have to be flushed out. Similarly a page-in results in a message to memory object port if the page is not resident in physical memory. Mach contains a *Default memory manager* that allocates zero filled memory regions. A task can set protection, inheritance characteristics of individual memory pages. Portions of the address space may be shared through inheritance or external memory management.

2.2 C Threads

The Mach kernel provides a set of low-level, language independent primitives for manipulating threads of control in support of multi-threaded programming. The C Threads package is a run-time library that provides a C language interface to these facilities [Loe91]. While using the low-level thread primitives allows complete control over threads, it entails handling machine dependent aspects such as setting up the machine state of the thread [B+93]. The usage of the C Threads package facilitates multi-threaded programming with machine independent function calls.

2.3 Lites

Lites is a 4.4 BSD based single server Unix emulation over Mach [Hel94]. Its organization is shown in Figure 2. It comprises of a server and an emulator library. The server provides the full BSD functionality such as processes, file systems, and sockets. The major change is in the way it interacts with the application. The server is an event driven multi-threaded program that responds to requests from applications. The emulator library is mapped into each application address space at a fixed virtual address at the time of *exec* system call. A Unix process is implemented by a Mach task with a single thread. When the application invokes a system call by trapping into kernel, the Mach kernel recognizes that the system call is being emulated and passes control back to a user location in the task. The emulator lies at this address. It decodes the request, communicates with the server, if required, and replies to the the application. Thus the application finds a seamless Unix environment, even though it is being provided by a server and an emulator. The server treats the emulator as just another shared library.

2.3.1 Client-Server Interface

The application communicates with the server through the emulator. The emulator uses different ways to talk to server. They include RPC, shared data, and page faults. The server shares some data with each process. It includes both read-only

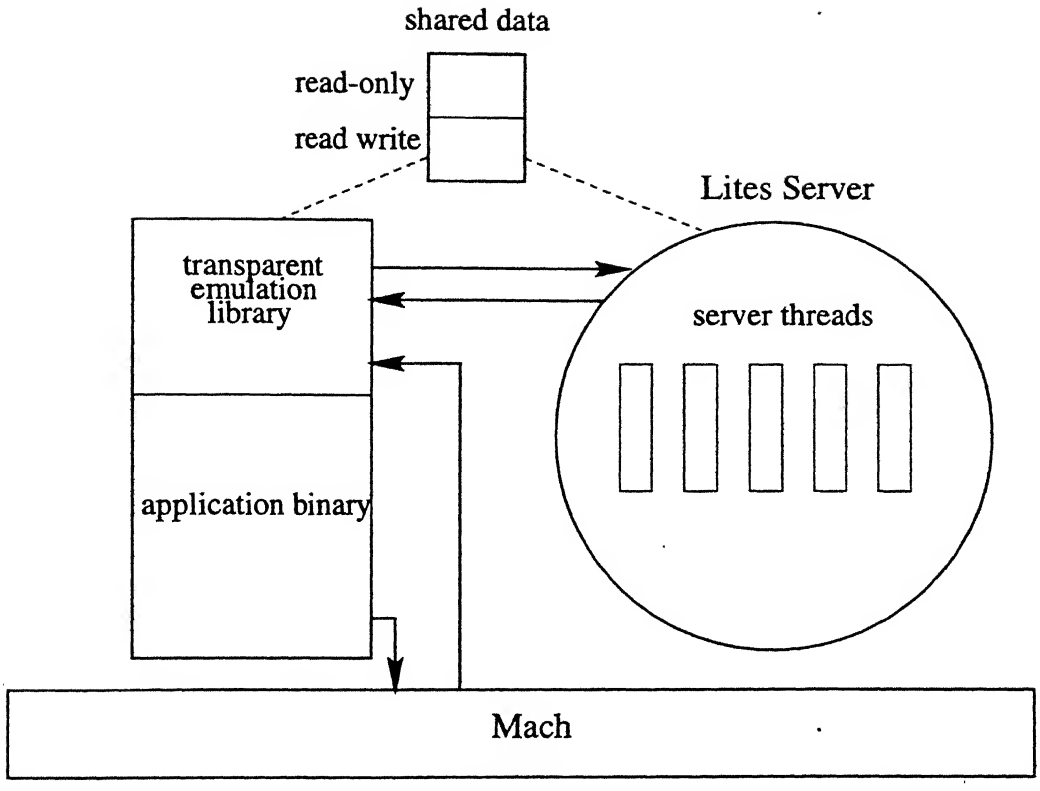


Figure 2: Lites Structure

data such as current signal and read-write data such as signal masks. The emulator can thus service some system calls (for example, *sigblock* to block signals from delivery) without sending a message to the server.

The Lites server also functions as an external pager for files. Executable files which may not be modified are mapped into application address space and the server services page-ins in this case. But, in general, the files which may be modified cannot be mapped directly as the server lacks the code for maintaining coherency among multiple simultaneous users of the same file.

The server is a passive entity that responds to application's requests in an event driven fashion. However, a notable exception to this scheme is in handling asynchronous signals which are not results of exceptions but those generated by the *kill* system call. In this case, the server sends a signal notification message to the emulator.

2.3.2 Concurrency Control

Lites derives most of its code from 4.4 BSD. But there are significant differences between the monolithic BSD and Lites. Lites is a parallel program, with multiple threads running to provide services. On a multiprocessor, these threads can actually run in parallel. In BSD, at any given time, the kernel has a single point of control. Also, as Lites is a user level program, it can be preempted at any time. The BSD kernel is divided into two halves. The top half executes in a process context and the bottom half, in the interrupt context [LMKQ89]. The only way the kernel is preempted, while in top half, is when an interrupt occurs. The top half can block interrupts by raising the processor level to protect data structures that are shared by both halves. Interrupts cannot occur in Lites. So it simulates them with an array of condition variables and mutex variables provided by the *cthreads* package [B+93]. Threads in Lites synchronize using mutexes.

2.3.3 Multi-threaded Clients

As Mach programs can be multi-threaded, Lites also allows the processes to be multi-threaded. This means that the process may make multiple system calls at the same time and the server should incorporate mechanisms to keep track of these multiple invocations. The traditional BSD kernel stores information about a process in a global *proc* structure. In Lites, this is split into two parts. A global process state — which is common to all invocations, and the state which is specific to particular system call invocation. A system call is handled all the way by a single thread. So this system call specific state is made thread specific and allocated on the thread stack. At the beginning of a system call, this structure is linked to the *proc* structure.

2.3.4 System Call Entry

When the emulator needs to communicate with the Lites server in order to service a system call, it makes an RPC on a port that represents the process in the server. The server pools all these ports and ports representing other objects in a port-set. Server threads that are not currently executing wait in a message receive operation on this

port-set. When a message arrives on a port, the generic server loop calls the MIG[1] generated stub routine which, depending on the request-id, calls the appropriate system call handler. The first thing that this routine does is to identify the internal object the port represents. This is handled by port object module explained below. It then takes certain locks and proceeds to execute just as in traditional kernel. Finally the loop replies to the request and again waits on the port-set. If the message request-id is not recognized, an error is returned.

2.3.5 External and Internal Objects

The Lites server has to keep track of many objects that it maintains such as processes, devices, and memory objects. These external objects are represented in the server as ports — receive rights or send rights. There is an internal data structure (internal object) corresponding to each such external object. An external object adds a reference to an internal object. For example, a *process* is represented by a receive right in the server and has a *proc* structure associated with it. The server should be able to identify the internal object, upon receiving a message from an external object.

Figure 3 shows all the objects in Lites. Circles stand for ports representing external objects while a rectangle represents an internal object. A circle with an incoming arrow stands for a send right and a circle with an outgoing arrow stands for a receive right [Hel94].

On the arrival of a message on a port, its corresponding internal object should be identified. The termination of external objects should result in decrementing the reference count of internal object and deallocating it when it reaches zero. Mach provides two kinds of notifications that can be used for this. There is a *no-senders-notification* for receive right which means that there are no more existing send rights for this port. For a send right, a *dead-name-notification* means that the corresponding receive right has been destroyed. Garbage collection can be carried out on receiving these notifications. These notifications are requested at the time of

[1]Mach Interface Generator produces client and server stubs from a specification file, to reduce the complexity of Mach IPC calls

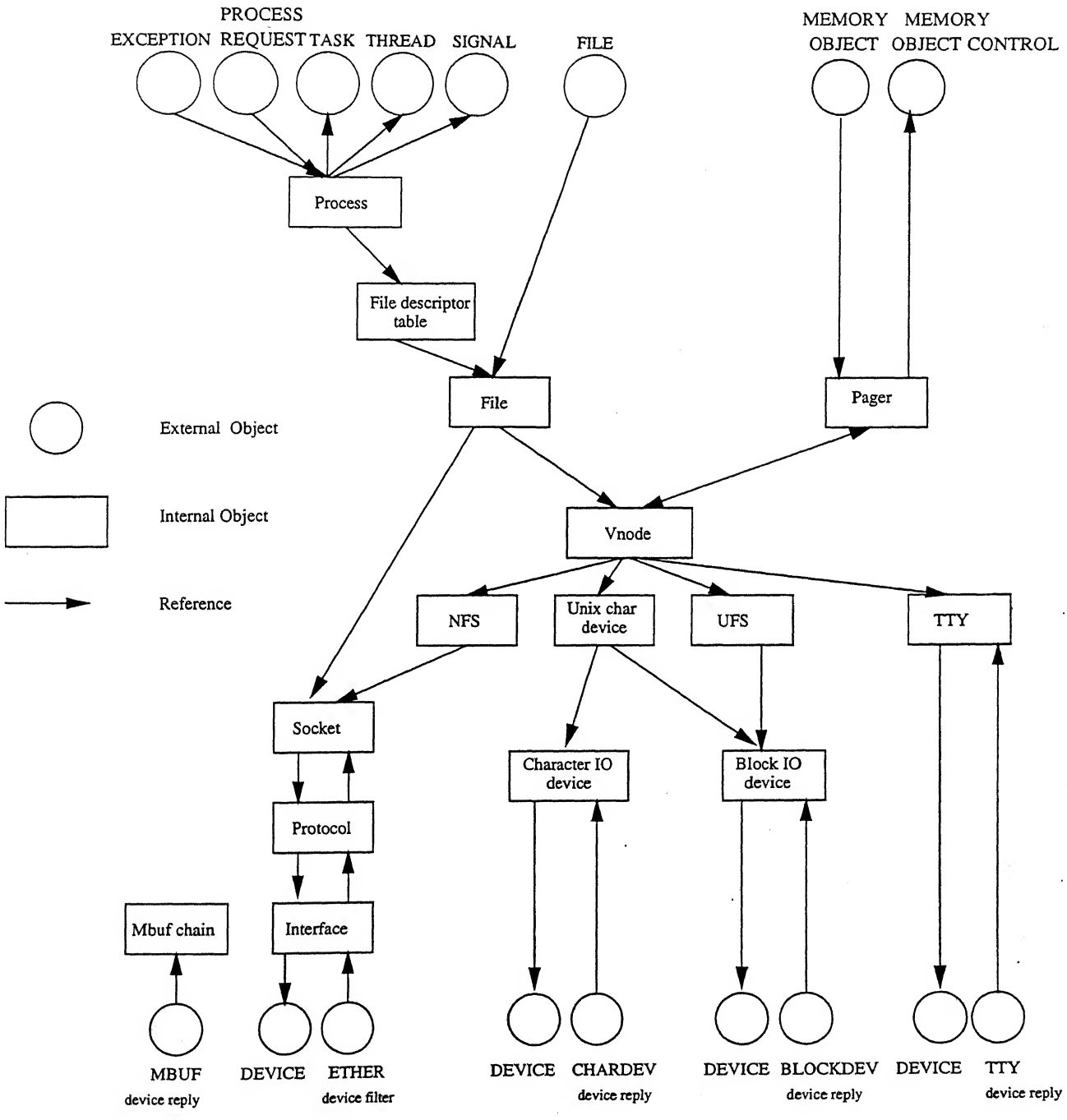


Figure 3: Server Objects

creation of respective ports.

To implement all the above operations on ports, the Lites server used a *port-object abstraction.* A port-object stores the type of the port, state of the port, and a pointer to the internal object. The port-object module is responsible for lookup, garbage collection, shutdown, serialization, and locking issues involved with ports.

Chapter 3

File-server Design

The previous two chapters detailed the background issues related to the present work. In this chapter, we describe various issues in the design of our file server.

3.1 Design Goals

As mentioned earlier, the monolithic structure of the traditional Unix kernel and of the single Unix emulation server lacks modularity. This makes it difficult to extend the system by adding new types of file systems. We want to structure the system so that adding new file system types will be relatively easy. Moreover, this addition should not require disrupting the existing system. Thus *modularity* was our major design goal.

In a modular architecture, logically separate components interact through a standard communication mechanism. This may result in some degraded performance as system calls are more efficient than communication through messages. We want to keep the number of extra messages thus generated to a minimum, and thereby ensure a reasonable performance of our system.

3.2 Single File-server or Multiple File-servers ?

Most Unix kernels support multiple types of file systems through a Virtual File System (VFS) layer which was first implemented in SunOS [SGK+85] in order to support NFS. VFS is an indirection layer that carries out file system type independent operations. The kernel calls the functions in the VFS layer for all the file system related system calls; the VFS layer, after carrying out some generic operations, calls the routines specific to an implementation of a file system type to complete the call.

In such a scenario, the major issue in file server design is: whether to have a single file server which implements all types of file systems or to have a separate file server for each type of file system. A single "universal" file server would be simple, but also monolithic, making the addition of new file systems difficult. This would defeat the very purpose of this project; we do not want a monolithic server any more than we want a monolithic kernel or a monolithic Unix emulation server! Thus we decided to have a different file server for each type of file system. In such an architecture, it is very easy to add support for new file system types. In fact, in our implementation, it is possible to add such support without even rebooting the system. Also, there is no need for the VFS layer code any more since each server implements just one file system type. But, in this architecture, the various file servers will have to cooperate among themselves in order to provide the illusion of a single large file name space. This cooperation will require that these servers communicate with (i.e, send messages to) each other probably resulting in degraded performance. In order to keep the performance at a reasonable level, we must make sure that such inter-server communication is as little as possible.

As mentioned earlier, in this phase of the project, we have just removed the file system functionality from the Lites server. Thus the original Lites server has been "thinned" down to provide all services other than the file system related ones. This new server is referred to as the Reduced Lites Server (RLS) in the rest of this thesis.

3.3 File system Data Structures

In Unix, a process holds three kinds of file references [Bac86]. It has a current directory, with respect to which the relative pathnames are resolved; it has a current root, which is used to resolve absolute pathnames, and finally, the other reference is through open files.

A process has to open a file to access the data in it. The *open* system call returns an integer called user file descriptor. Other file operations, such as reading, writing, setting file I/O parameters, and closing the file, use the file descriptor that the *open* system call returns. The user file descriptor is an index into a table in the kernel called user file descriptor table, which is unique to every process.

The kernel internally represents a file by an unique inode. An inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file data in the file system. Thus all operations on a file require manipulating its inode. The kernel converts the file name in a system call to its inode, and keeps the inode in the inode table.

At the time of opening a file, the Unix kernel allocates an entry in the global file table. The file table entry contains a pointer to the inode of the open file and an offset value in the file where the kernel expects next *read* or *write* to begin. The entry in the user file descriptor table points to the entry in the global file table. The relationship between the user file descriptor table, file table, and the inode table data structures is shown in the Figure 4.

The file table entries of all instances of an open file point to one entry in the inode table. The user file descriptor table could conceivably contain the offset for the next I/O operation and point directly to the entry in the inode table, eliminating the need for a separate kernel file table. But the file table is implemented as a separate data structure in order to allow sharing of the offset field by several file descriptors; the system calls *dup* and *fork* allow such sharing. When a new process is created using the *fork* system call, the child process inherits the user file descriptor table from its parent process, thus allowing it to access the files opened by its parent. Immediately after the fork operation, the corresponding user file descriptor table entries of both the parent and the child point to the same file table entries, thus

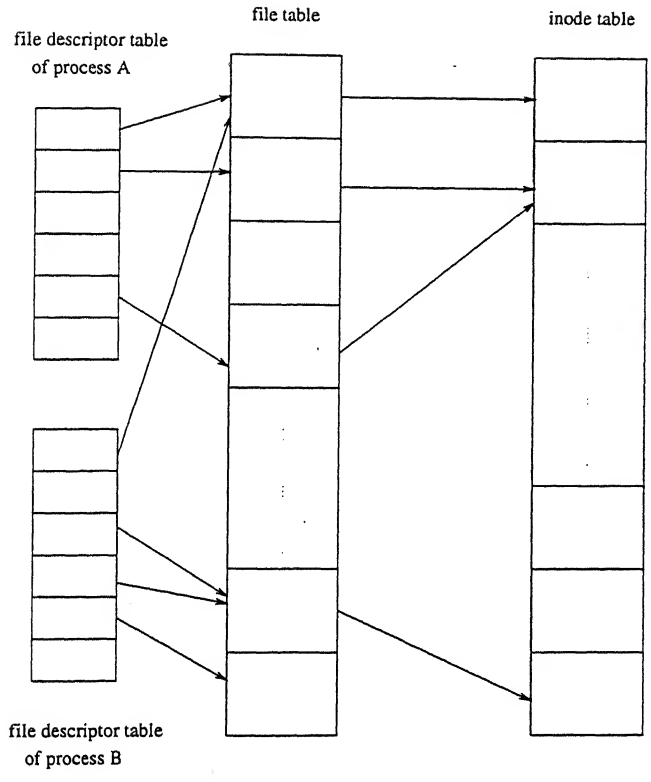


Figure 4: Unix file system data structures

sharing the offset field. After fork, however, opening a file in one process is not reflected in another.

In Unix, a file descriptor can also refer to a socket. The *socket* system call returns this descriptor, which is used in other operations on the sockets, such as sending and receiving the messages. For socket descriptors, the file descriptor table entry points to a socket data structure, instead of a file table entry. Some system calls such as *read* and *write* work on both kinds of descriptors.

In our design, a file server implements a single type of file system. Typically, multiple file servers may be simultaneously running in order to support different types of file systems. It is not possible to have the file descriptor table of a process in any particular file server, since a process can simultaneously have open files belonging to several different file system types. RLS can maintain this table, but in that case, the operations on file descriptors result in at least two messages — from the emulator

to RLS, and from RLS to a file server. Since reads and writes on a file constitute majority of file operations, we do not want the operations on file descriptors to result in many messages. In such a scenario, the emulator is the correct place to keep the file descriptor table. Similarly the emulator should also hold the other file references — current directory and the current root.

In order that the operations on the file references are efficiently implemented, we represent files and directories with ports. As mentioned earlier, a file table entry contains an offset value for the next I/O operation in the file, which is shared between different file descriptors; this requires that the file table is in the file servers. An entry is allocated in the file table for every open file, just as in the traditional Unix kernel. But we also allocate file table entries for directories, for the sake of uniformity. A port is associated with each file table entry and the send right to it is returned in system calls such as *open*, *chdir*, and *chroot*. Thus the emulator has send rights for current directory, current root, and all open files. The location of and the relationship between the file descriptor table, file table, and the inode table in our design is shown in the Figure 5. For the sake of brevity, only one process and one file server are shown in the figure. The dotted lines indicate the association between the send rights and their corresponding receive rights.

As already mentioned, we have only removed the file system functionality from the Lites server. Therefore, in the present implementation, socket services are being provided by RLS. It maintains a descriptor table for each process for this purpose. For a socket descriptor, the emulator stores the socket's index into this table in RLS, in its own file descriptor table entry; the descriptor which the application gets is the index into the descriptor table in the emulator. In all the operations on socket descriptors such as *send* and *receive*, emulator replaces the descriptor parameter with the integer stored in its descriptor table entry. Thus the emulator, in its descriptor table, stores a send right for a file descriptor, and an integer for a socket descriptor.

3.3.1 Shared State between emulator and RLS

In Unix, the *fork* system call is the only way to create a new process. The new process inherits most of its process state from its parent. The inherited state includes

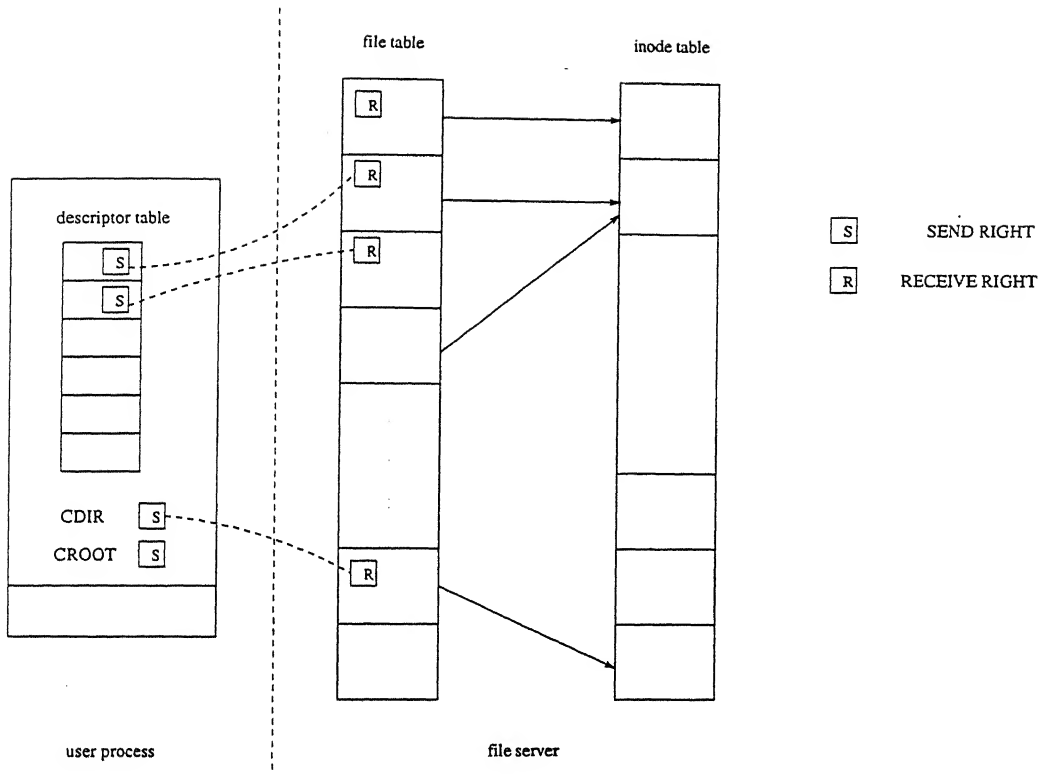


Figure 5: file system state in the new design

the current directory, current root, umask, and the file descriptor table. As already explained, this state is present in the emulator in our design. As the *fork* system call is implemented by RLS, it should be able to access this data in the emulator in order to copy it: To make this copying easier, RLS shares this data with the emulator.

3.4 Mounting a File system

A file system, in Unix, is characterized by a hierarchical structure with a single root node. The user's notion of a single large file name space is built by attaching multiple file systems at different directories. This process is known as mounting, and the directory on which the file system is mounted is called a mount point. A mount table maintains the mappings between the mount points and the root directories for

all the active file systems (file systems which are mounted). This allows the Unix kernel to cross file systems transparently while parsing the pathnames, creating the illusion of a single file name space. Since, as explained earlier, Unix supports different types of file systems, it is possible that the mount point lies in a file system of a type different than that of the mounted file system. The mount table entry also contains some file system specific information such as superblock, flags, and statistics.

Figure 6 shows an example of mounting a file system. In the figure, the special

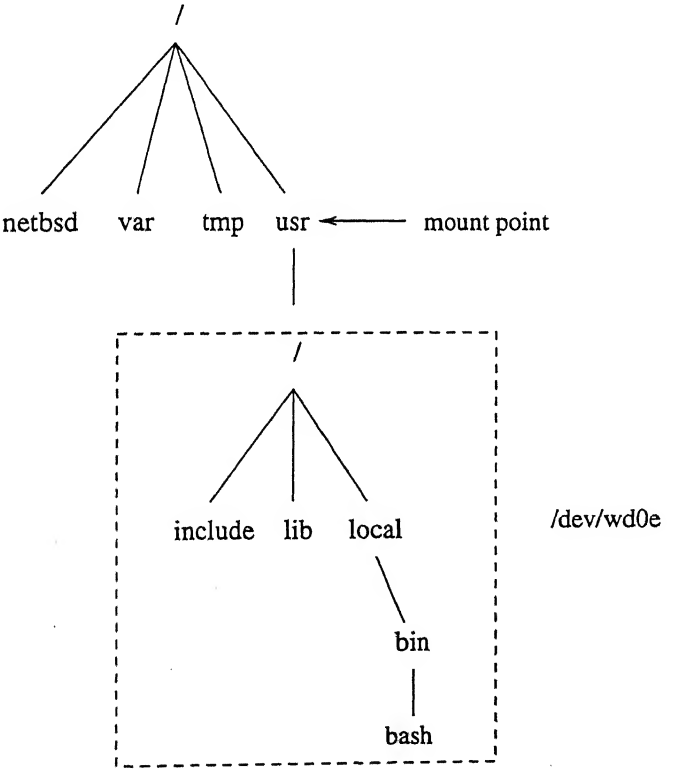


Figure 6: An example for Mount

device file "/dev/wd0e" contains a file system. This file can be opened as a special device file and the data in it can be accessed just as in a regular file. But the mount operation allows it to be accessed as a file system. In order to mount this file system, the Unix kernel opens the block device; it reads the superblock and the root inode of the file system, and gets the inode of the mount point "/usr". A mount table entry

is created, and the mapping between the mount point inode and the root inode is established; some other information including the superblock is also stored in this entry. The root node of this file system is now designated as "/usr". The processes, henceforth, can carry out file operations on the file "/usr/local/bin/bash" without needing to know that the file actually exists on a separate file system.

As already explained, a mount point may lie in a file system of a type different than that of the mounted file system. In our design, this implies that the mounted file system and the file system containing the mount point are managed by two different file servers. The mount processing, such as opening the device and reading the superblock, is carried out by the file server that manages the mounted file system. In such a scenario, the mapping between the mount point and the root directory of the mounted file system has to be maintained across the two servers. In order to achieve this, we split the traditional mount table into two separate tables; mount table and manage-fs table. The association is now maintained between the ports representing the mount point and the root directory of the mounted file system. Thus the mount table entry maintains the mapping between mount point port and the port for the root directory of the mounted file system; the corresponding manage-fs table entry also contains the above mapping in addition to the file system specific data such as the superblock.

The associations in both the tables allow the file servers to communicate with each other to create the illusion of a single large file name space, as explained in the following section. Thus corresponding to every active file system, there are two file servers. One manages the file system containing the mount point and contains the mount table entry for this mount; the other file server manages the mounted file system and contains the corresponding manage-fs table entry. These servers may be the same if mount point belongs to the same type of file system as that of the mounted one.

We need not start file servers for all types of file systems supported by the system at boot time. We only need the file server which manages the root file system. Once the system is booted, we can run file servers for different types of file systems when the first request for mounting corresponding type of file system is received. For

example, if the root file system is of type FFS, the FFS server is started at boot time and it manages the root file system. After some time, if there is a request to mount a file system of type X, RLS locates the corresponding file server image and starts it. The executable images of all the file servers should be present in the directory "/mach_servers", which itself must be present in the root file system. The name of a file server should be of the form "fs_*fs-type*" where *fs-type* is as given in "sys/mount.h". For example, the executable image of the file server for FFS will be "/mach_servers/fs_ufs". Thus new types of file systems can be added without needing to reboot the system.

3.5 Resolving the Pathname

As mentioned before, the Unix kernel internally represents a file by an inode: It converts the file name specified in the system calls to the corresponding inode by parsing the pathname one component at a time: It checks that all the intermediate components are directories, and that the process has required permissions to search them; it eventually retrieves the inode of the file, and can access the file data from the information available in the inode. The kernel resolves the pathname with respect to the current directory of the process, if the pathname is relative and, if the pathname is absolute, the current root of the process is used as the starting directory to resolve the pathname.

While parsing a pathname, the kernel may encounter a component which is a mount point. Further parsing of the pathname should be carried out with respect to the root directory of the file system that is mounted on this directory. The kernel obtains this root directory inode from the corresponding mount table entry and continues the parsing. Similarly, if the path so far resolved is the root directory of a mounted file system, and the next component in the pathname is "..", further parsing should be done with respect to the mount point inode as the ".." refers to the parent directory of the mount point. The kernel gets the the mount point inode from the corresponding mount table entry, and proceeds to resolve the pathname with respect to the mount point. Thus the kernel resolves the pathnames that span

different file systems in a transparent manner, providing the illusion of the single large file name space to the user processes.

In our design, as mentioned before, the emulator holds ports for the current directory and the current root of a process. It sends the system calls containing pathnames such as *open* onto current directory port or the current root port, depending on whether the pathname is relative or absolute. A file server can, thus, obtain the starting directory, for resolving the pathname, from the port on which the message is received; it then proceeds in the usual manner except in the following three cases.

- A component in the pathname is mount point. As explained before, further parsing should be carried out with respect to the root directory of the mounted file system, and the root directory may not be in the same server. So the message is forwarded to the root directory port of the mounted file system; this port is obtained from the mount table.

- The path resolved so far is the root directory of a file system, and the next component in the path is '..'. This '..' refers to the parent directory of the mount point. So the message is forwarded to the mount point directory port. This port is obtained from the manage-fs table. (Note that, in the above two cases, if the two file system types are the same, then the file server forwards the message to itself).

- A component is symbolic link and the link contains an absolute pathname. In this case, the message is forwarded to the root directory port of the process. This port is available in the message itself. The emulator sends the root directory port in all the system call messages containg pathnames.

Thus the pathname in a message may not be resolved completely in the file server which received it initially, and the message may need to be forwarded. This forwarded message contains the pathname which is yet to be resolved. Eventually the message reaches a file server which is managing the file system containing the file or directory that is referred to by the pathname, and thus, the pathname can be

resolved to its inode. This server carries out the requested operation and directly replies to the emulator which is unaware of the forwarding that may have taken place.

The forwarding of system call messages is necessary only for those calls that contain pathnames spanning different file systems. Those calls that work on file descriptors are sent directly to the port representing the corresponding file, and thus, need no forwarding. Thus even though the number of messages for a system call containing a pathname may be relatively more depending on how many file systems are crossed, the system calls that work on file descriptors still result in only one message. Since *reads* and *writes* constitute majority of file operations, and they need only one message, we expect the performance of new architecture to be reasonable.

Figure 7 shows the system structure when two file servers are running. The root file system is of type FFS (Fast File system). An Ext2 file system is mounted on "/usr". The mount table entry in FFS server contains the port representing the root directory of Ext2 file system. Similarly the manage-fs table entry in Ext2 server shows the send right for the mount point. The figure also shows two application processes with all the relevant data structures — file descriptor table, current directory, and current root. The association between the send rights and their corresponding receive rights is indicated with dotted lines. Assume that the application 1 invoked open system call on the file "/usr/local/bin/x". The emulator in application 1 then sends the open message to the current root, as the pathname is absolute. As shown in the figure, the current root port of the application 1 is in FFS server, and thus, FFS server receives the open request. While parsing the pathname, the FFS server finds out that "/usr" is a mount point; it then forwards the message to the root directory of Ext2 file system, which is in the Ext2 server. The forwarded message contains the pathname "local/bin/x". As this file is in the mounted file system, Ext2 server resolves the pathname to its inode, carries out the open system call, and replies directly to the application 1. The emulator is not aware of the forwarding that took place. On the other hand, when the application 1 invokes open on the file "/tmp/f", the open message is sent to the current root again; the FFS server

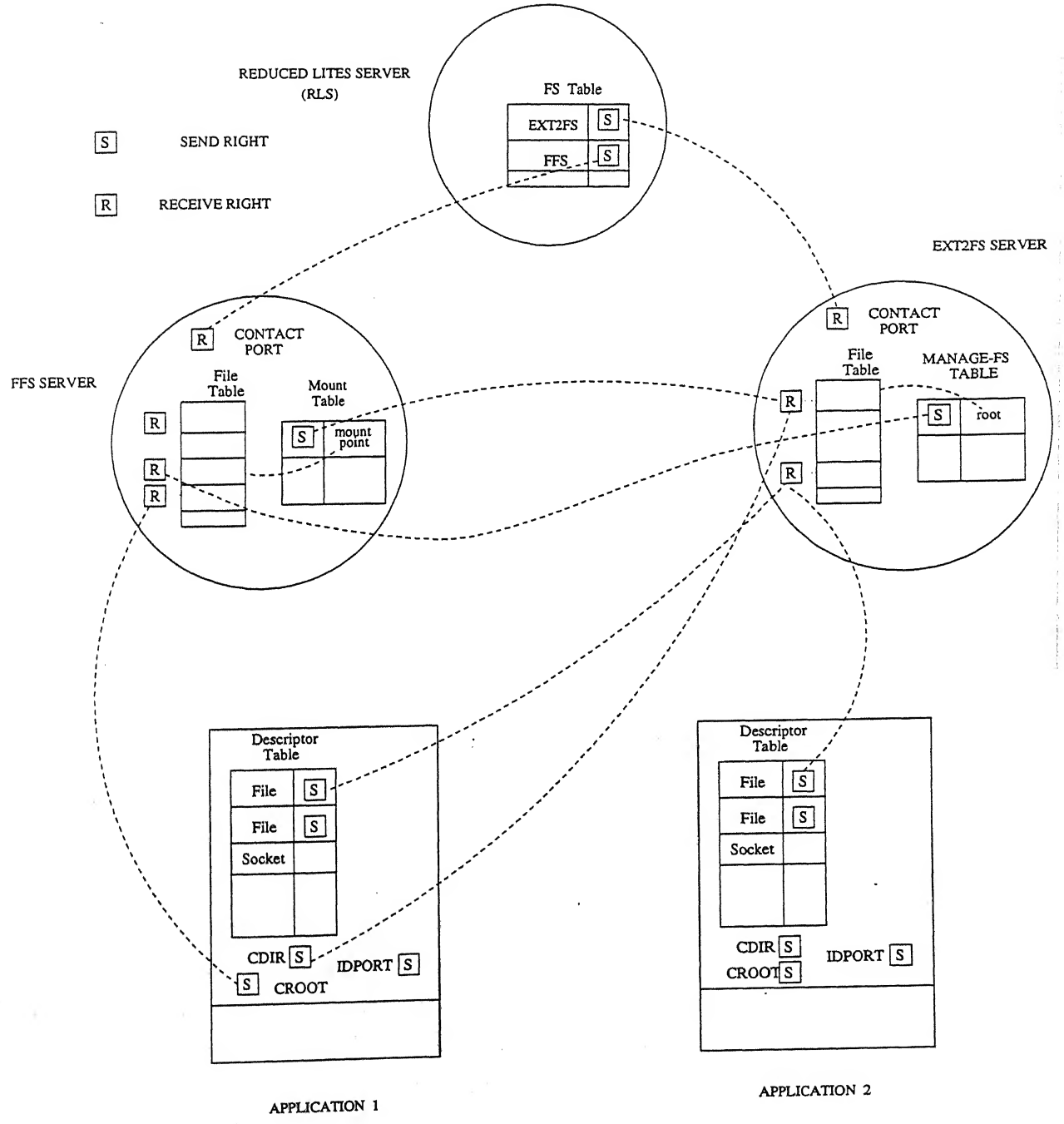


Figure 7: LitesNew structure

that receives this message will be able to resolve the pathname completely, and reply to the application 1 without any forwarding since the pathname "/tmp/f" is contained entirely within one file system.

3.6 Interaction between RLS and file servers

At times, the file servers and RLS need to communicate with each other to provide services to the applications.

The file servers do not maintain any process specific state. But at times they need to use some process context information. For example, permission to access a file is controlled by access permissions associated with the file; to check these permissions of a file, at the time of opening, the file servers require process credentials — UID and GIDs. The emulator can send this data in the message, but it can not be trusted, as in that case, it can easily masquerade as another user with more privileges than that of itself. In order to have a secure mechanism of getting the process specific data, a port, called idport, is associated with each process by RLS, and a send right to this port is made available to the emulator. The emulator sends the idport in those messages to the file servers that may require to use the process context in their implementation, such as *open*. Thus the file servers can get the process credentials by sending a message to RLS, identifying the process by its idport.

The emulator cannot forge a send right of some other process's idport, and thereby claim its privileges. This is ensured by the Mach kernel as the port rights in Mach are kernel protected entities. But it is possible for a process to pass the send right to its idport to another process under some circumstances. In order to insert a port right in a task, its task control port is required. A process can obtain the task control port of another process if it is a privileged process or if both the processes have the same user ID; in such a case, the process can insert the send right to its idport in the second process. The second process can send this idport in the messages to the file servers, and consequently claim the first process's privileges. But this is not really a security violation as this masquerading can happen only if a process can obtain the task control port of another process, and it can obtain this

port only if it is authorized to do so. There is no other way for a process to obtain the task control port of another process. Thus the idport mechanism is a secure way of obtaining the process context information in the file servers.

Another occasion when the file servers need to contact RLS is related to signals. Certain characters typed at a terminal result in a signal being sent to the *foreground process group* of the terminal. In order to send a signal to a process or a process group, the file servers send a message to RLS specifying the corresponding process or process group and the signal to be sent.

Chapter 4

Implementation of the FFS Server

In the last chapter, we described the design of our file server architecture. As has been seen, our design supports all types of file systems. However, currently we have only implemented the file server for Fast File System (FFS) which is the native file system on BSD [LMKQ89] systems. In this chapter, we discuss the issues related to this implementation.

4.1 Starting up File servers

As explained earlier, we run a file server only when a request for mounting the corresponding type of file system arrives.

The file servers have a port called the contact port, which is used to receive the requests for managing a device in order to mount a file system in it. This port is created immediately after a file server starts running. When a Unix system is booted, at first, a process is "hand-crafted". This is process 0, which always runs in the kernel mode. In Lites, the process 0 is the server itself, and just like all the other processes, it has a port. When a file server is started up, RLS makes the process 0 port as the bootstrap port of the file server. As has been seen, the file servers need to communicate with RLS, and they do this by sending messages to their bootstrap ports.

The system structure with the relevant data structures, when two file servers

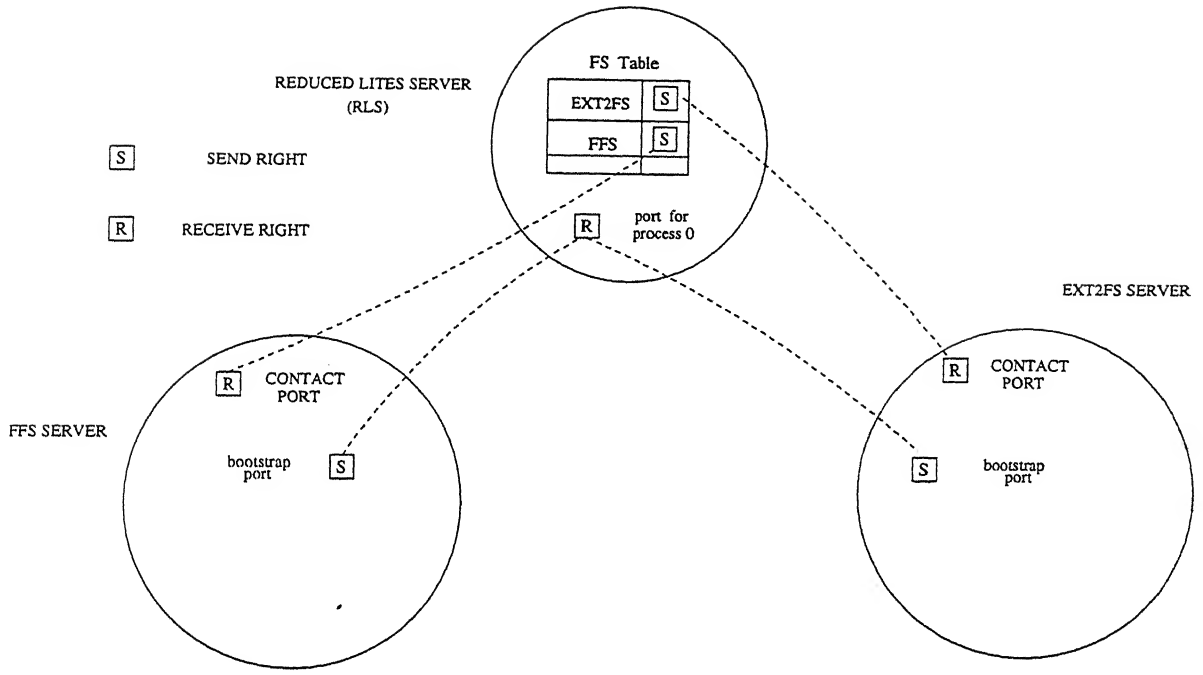


Figure 8: Association between RLS and file servers

are running, is shown in Figure 8. RLS has a table *FS_Table* which maintains the mappings between the file server types and their contact ports. When a mount request for a file system arrives, RLS checks if it has an entry for the corresponding type in *FS_Table*, and if not, it starts up the file server. The file server, on coming up, creates the contact port and sends a message to RLS containing the send right to its contact port, and thus, enabling RLS to establish the mapping.

4.2 Shared state between RLS and Emulator

As mentioned before, the Lites server shares some process context with each process, enabling the emulator to service some system calls without sending a message to the server. The shared state consists of read-only data and read-write data. In our design, as has been seen, there is a third structure that is shared.

When a process is created, RLS maps the shared object at a fixed location

EMULATOR_END in the process address space. The shared state consists of three pages as shown in the Figure 9; the structure *ushared_ro* designates the read-only state while *ushared_rw* is the read-write state. These two structures are same as those in the Lites server.

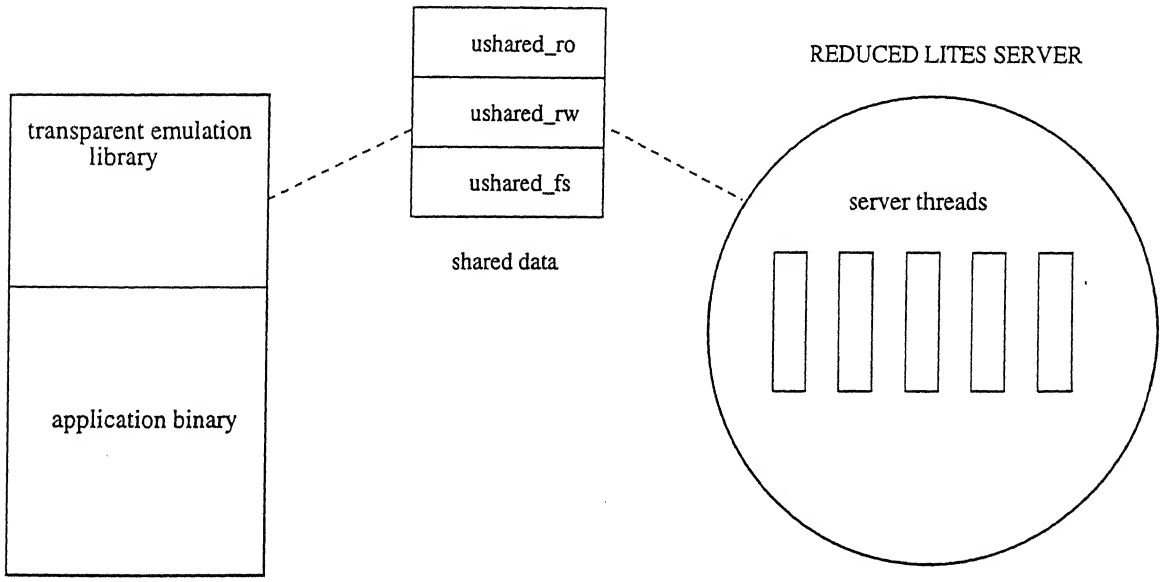


Figure 9: Shared state

The *ushared_fs* structure is the process state related to file services that is moved to the emulator in the new architecture. It consists of the descriptor table, the current directory (cdir), the current root (croot), and the umask. This state also comprises of the idport of the process.

While creating a new process, RLS copies this state (except the idport) to the child process. But mere copying the data does not bestow the child process with the send rights for the current directory, current root, and the ports for open files. The port rights have to be explicitly extracted from the parent process and inserted in the child. RLS has the task control ports for both the parent and the child processes, and thus, carries out this extracting and inserting of port rights with the calls *mach_port_extract_right* and *mach_port_insert_right*.

The idport field in the *ushared_fs* structure is the port name of the process idport. The actual send right is explicitly inserted by RLS.

4.3 Mounting a File system

In the previous chapter, we explained the high level details of mounting a file system, and how the mount table is split into two parts. In this section, the low level details related to the mounting are described.

The emulator determines the device number corresponding to the special device file containg the file system to be mounted, by sending a "stat" message for the file. It then sends a mount request to RLS containing this device number, in addition to other parameters. RLS appends the contact port of the file server corresponding to the type of the file system to be mounted (Note that if the required file server is not running, RLS starts it, as explained earlier), and forwards the message to the current directory or current root of the process depending on whether the mount point pathname is relative or absolute. The ports for current directory or current root can be obtained by extracting them from the process. Thus the message reaches the file server which is managing the file system that contains the mount point, possibly after crossing different file systems. This file server creates a mount table entry for this mount, and the port for the mount point directory. It then sends a *manage_fs* message to the contact port that is present in the mount message; the *manage_fs* message consists of the port for the mount point and the device number containing the file system to be mounted. As can be seen, if the file system to be mounted is of the same type as that corresponding to this file server, it sends the *manage_fs* message to itself.

The file server which receives the *manage_fs* message carries out the main mount processing: It opens the device, reads the superblock and the root inode, and creates a port for the root directory. It then creates a manage-fs table entry and establishes the mapping between ports for the mount point and the root directory of the mounted file system, and finally sends a reply to the *manage_fs* message.

This reply message contains the port for the root directory of the mounted file system, enabling the file server that sent the *manage_fs* message to establish the above mapping in the mount table.

Thus the file server that manages the file system containing the mount point completes the mount call, and replies to the emulator.

4.4 Unmounting a File system

In Unix, the *unmount* system call disassociates the mounted file system from the mount point directory. The dirty buffers belonging to this file system, and the superblock are written to the disk. The device containing the file system, which is opened at the time of the mount, is closed. Finally, the corresponding mount table entry is removed.

The *unmount* system call takes the mount point pathname as a parameter to specify the file system to be unmounted. As already explained, the emulator sends the unmount message to the current directory or current root of the process depending on whether the pathname is relative or absolute. This message reaches the file server that is managing the mounted file system, where the mount point pathname is resolved to the root inode. This file server carries out the unmount processing such as writing the superblock to the disk. It removes the corresponding manage-fs table entry after deallocating the send right for the mount point. Since this is the only reference to the mount point port, its deallocation causes the Mach kernel to send a *no_senders_notification* to the file server holding the receive right for the mount point. This server, upon receiving the notification, destroys the port, and removes the corresponding mount table entry.

4.5 Port Objects

The FFS server has to keep track of many external objects such as files, directories, devices, and memory objects. They are represented in the server by a send right or a receive right; a file, for example, is represented by a receive right. There is

an internal object in the server corresponding to each such external object. For example, the internal object for an open file is the corresponding file table entry. Upon receiving a message on a port, its corresponding internal object should be identified before carrying out the requested operation. An external object adds a reference to the corresponding internal object, and its removal, therefore, should decrement the reference count of it. The internal object will be deallocated if its reference count reaches 0.

These operations of lookup and garbage collection are implemented using *port-object* module. A port-object stores the type of the port-object and a pointer to the associated internal object. Immediately after receiving a message on a port, its corresponding port-object should be identified. This is done differently for send rights and receive rights. A hash table is maintained for send rights to find out their corresponding port-objects, while for receive rights, finding out their corresponding port-objects is more efficient. The port name of the receive right is made same as the address of the port-object structure, and thus simple type casting of this port name gives the associated port-object. The port-object lookup routine returns the pointer to corresponding internal object after calling the routine POT_LOCK on it. Similarly on receiving a *dead_name notification* for a send right or *no_senders notification* for a receive right, the port-object module calls the routine POT_DEREFERENCE on the internal object. These notifications are requested at the time of creation of the ports. The functions POT_LOCK and POT_DEREFERENCE are indirect function pointers, and are statically set for each type of port-object at compile time.

Figure 10 shows all the objects in the FFS server. The circles stand for the ports representing external objects while a rectangle represents an internal object. A circle with an incoming arrow stands for a send right and a circle with an outgoing arrow stands for a receive right. All the objects shown in the Figure 10 are same as in the original Lites server except the FILE-DIR port-objects.

The FILE-DIR port-objects correspond to the ports representing files and directories. While creating a file table entry for a file or directory, the FFS server creates this port-object with the call *port_object_allocate_receive*. This routine allocates a port-object, and records its type as FILE-DIR; it creates a port with the same name

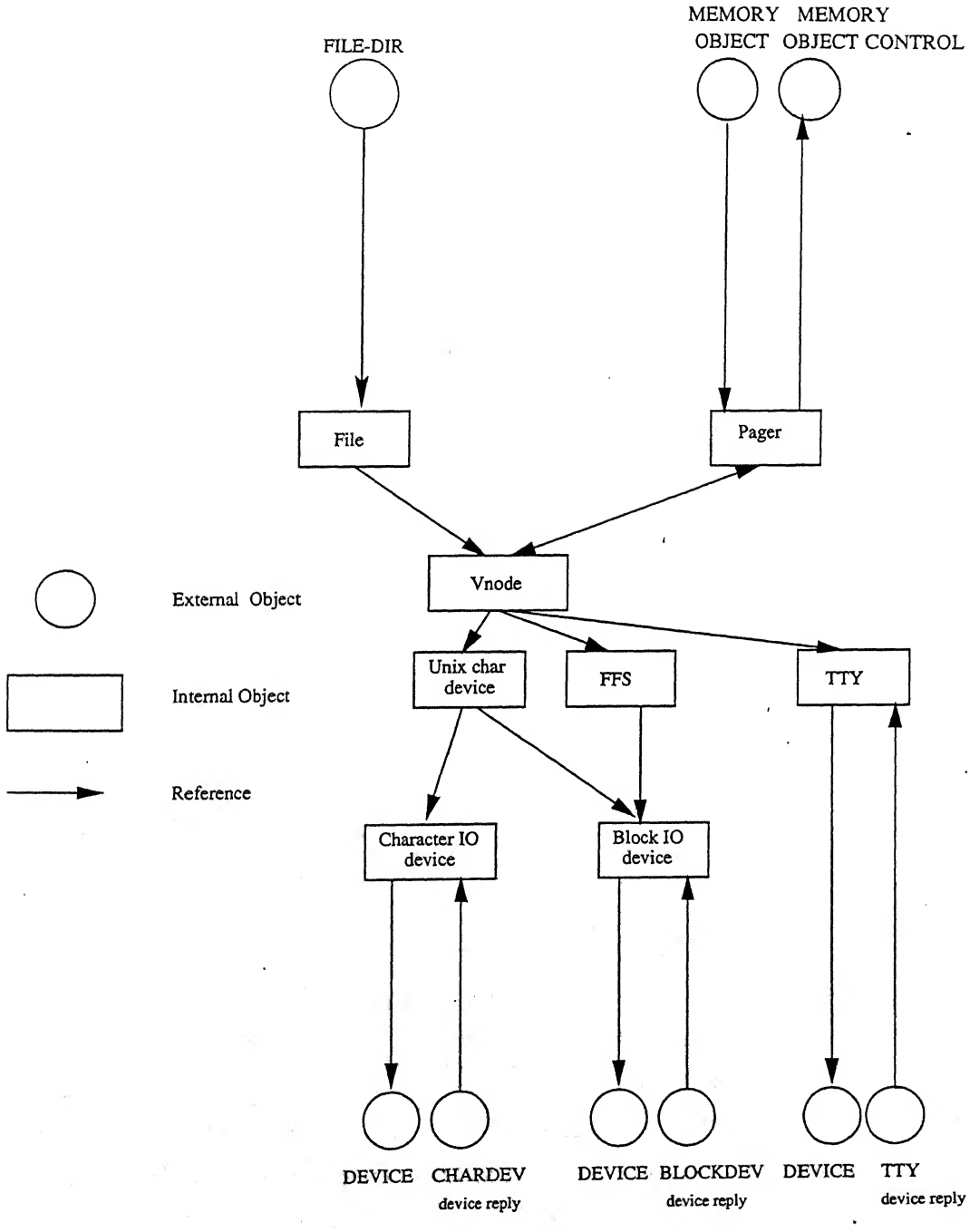


Figure 10: FFSobjects

as the address of the allocated port-object. The port-object contains a pointer to the associated file table entry. We request the *no_senders_notification* for this port, whose arrival will be used to delete the file table entry.

4.6 Controlling Terminal

In BSD, a process belongs to a process group, and some process groups together constitute a *session* which may have a *controlling terminal.* Certain characters typed at this terminal result in a signal being sent to a process group in the associated session. This process group is the *foreground job* of the terminal, and all the other process groups in the session are *background jobs* [LMKQ89].

The association between a session and its controlling terminal is maintained by storing pointers in session and tty structures to each other. But in the new architecture, it is not possible to store these pointers as the two structures are in different servers. While the session structure is in RLS, the tty structure is in a file server. In order to keep the association between these two structures in the new architecture, a port is associated with each session and each tty structure; the association is now maintained by storing send rights in session and tty structures to each other's corresponding ports.

4.7 Select

In Unix, the *select* system call is used to multiplex I/O on multiple descriptors. Three independent sets of descriptors, specified by masks, are watched for a change in their status. One mask specifies the descriptors for which the kernel checks if read will be possible without blocking. Similarly the second mask specifies the descriptors for which the kernel checks if write will be possible without blocking, and finally the third mask of descriptors is watched for exceptions. The call has a timeval parameter which specifies, if no descriptors are currently ready, the time interval for which the kernel should wait for one to become ready, before timing out. If the timeval parameter points to a zero-valued structure, the result is a poll of all the

descriptors. On return from the select call, the masks indicate the descriptors which are actually ready for the I/O.

In the BSD implementation of select, the kernel checks if I/O is possible, for each descriptor in each mask. If not, the fact that a process is selecting on the descriptor is noted in the internal data structures. If one or more descriptors are ready, the kernel returns after changing the masks to indicate the descriptors for which I/O is possible; otherwise it goes to sleep. It will be woken up either by timeout, or when a descriptor is ready for I/O. After waking up from the sleep, the kernel checks all the descriptors and returns after accordingly modifying the masks.

In our implementation, as mentioned earlier, a process can have open files belonging to several different file systems. Consequently, it is not possible to send the descriptor masks in select system call, as is done earlier, to any one particular server. In the new implementation of select, the emulator sends a select message for each descriptor: For file descriptors, this message is sent to the corresponding file port, while for socket descriptors, the message goes to RLS. Since the select handler in the file servers or RLS have the same functionality, any reference to a server in the following discussion can mean a file server or RLS depending on the type of the descriptor.

The emulator constructs a table of descriptors from the masks specified in the system call, with each entry in the table consisting of corresponding read, write, and except bits taken from the three masks. It then sends a select message for each descriptor with the corresponding read, write, and except bits set in the masks. The select message has a boolean parameter "poll" which specifies whether the server should just poll the descriptor or should block indefinitely waiting for the descriptor to become ready (the timeout mechanism is handled in the emulator itself, as explained below).

The select handler in the server checks if I/O is possible for the descriptor, and if so, it replies after accordingly changing the masks. The emulator stores these masks in the corresponding entry for the descriptor in the table. If I/O is not possible, however, the behaviour depends on the "poll" parameter. If it is true, then nothing more needs to be done. Otherwise, the server creates a port and a send right to

this port is sent in the reply; the select handler sleeps indefinitely in this case after sending the reply. The emulator can know whether I/O for a descriptor is possible or not, by checking whether the reply message contains a send right. The descriptor is ready if no send right is present in the reply. Otherwise, the descriptor is not ready, and the emulator stores the send right in the corresponding entry in the table that it earlier constructed. This send right identifies the thread in the server that is handling the select for this descriptor and is sleeping, waiting for the descriptor to be ready. When a descriptor becomes ready, the select handler in the server sends a message to the emulator. For this purpose, the emulator creates a port and sends the send right to this port in all the select messages.

Thus the emulator sends the select messages for all the descriptors, and in reply, it gets either a send right, specifying that the server thread is sleeping, or three bits indicating the status of I/O on the descriptor. It stores them accordingly in the entries for these descriptors in the table constructed earlier. If this is a poll, the emulator combines masks for all the descriptors, and returns after changing the input masks in place accordingly. Otherwise, the emulator checks if one or more descriptors are ready, and if so, it should interrupt the rest of the select handlers, causing them to wakeup from their sleep. In order to do this, it sends an interrupt message for all these descriptors, specifying in each case, the send right that identifies the corresponding select handler thread in the server. The server identifies the sleeping thread with the help of this send right, and wakes it up, which just returns without sending any message. Thus the emulator returns, after combining the masks for all the descriptors, and changing the input masks accordingly.

If no descriptors are ready, the emulator waits in the *mach_msg* call on the port whose send right was sent in all the select messages. The timeout parameter in the *mach_msg* call is set to the time interval specified by the timeval parameter in the select call. The *mach_msg* call returns either if the timeout expires or if a message is received from any of the servers. In case of timeout, the emulator sends an interrupt message for all the descriptors, indicating that the timer has expired. As mentioned earlier, the server threads wake up from their sleep and return without sending any message. Thus the emulator returns from select, indicating timeout.

If a message arrives from a server when the emulator is waiting in *mach_msg*, indicating that a descriptor is ready, the emulator stores the masks in the corresponding entry for this descriptor. It then sends an interrupt message for all the descriptors (Note that, there is no need to send the interrupt message for the descriptor for which the emulator received the reply. But as will be seen below, this is done to avoid a race condition, and the server which received this interrupt message behaves differently from others). On receiving the interrupt message, the select handlers in the servers wake up from sleep and check if the descriptor is ready. They send a message to the emulator indicating whether the descriptor is ready or not. The emulator waits for all the messages to arrive, and with each received message, it updates the masks in the table for the corresponding descriptor. Finally, it combines all the masks, and returns from the select call, after modifying the input masks in place.

As can be seen, there is a race condition between an interrupt message from the emulator, and the message from the server indicating that the descriptor is ready. For example, assume that the emulator sent select messages to two different servers S1 and S2 for descriptors fd1 and fd2, and is waiting in the *mach_msg* call. If fd1 becomes ready, S1 sends a message to the emulator. After receiving this message, as explained earlier, the emulator sends an interrupt message to S2. But, if at about the same time, fd2 becomes ready, then S2 also sends a message to the emulator. The interrupt message to S2 will not find the select handler since it returned after sending the message to the emulator. In order to avoid this race, S2 will not remove the map between the send right and the select handler thread after sending the message, but just notes that this mapping is now "passive" (i.e, there is no active thread for this send right). After receiving the interrupt message from the emulator, S2 checks if the mapping is "passive", and if so, it is enough to remove the mapping. There is no need to wake up the thread as the thread returned earlier. If the mapping is not "passive", however, it wakes up the thread as mentioned before.

4.8 Some Other Modified System Calls

4.8.1 Chdir and Chroot

The *chdir* system call changes the current directory of a process, while *chroot* changes the current root. In our implementation, the reply to the chdir message contains a send right to the port for the new directory. The emulator deallocates the send right for the previous current directory and starts using the new port right as its current directory. The *chroot* system call also follows similar pattern, with the reply message containing the port right for the new current root.

4.8.2 Dup and Close

In Unix, The *dup* system call duplicates an existing file descriptor to return a new descriptor. Both descriptors, old and new, refer to the same file table entry in the kernel. The *close* system call deletes the file descriptor from the user file descriptor table. If this is the last reference to the corresponding file table entry, it will be removed. These system calls work on both file and socket descriptors.

In our implementation, there is no need to send any messages for these system calls on the file descriptors. In order to implement *dup*, the emulator simply makes another send right from the one that it already holds. It allocates an entry in the descriptor table, puts the new send right in it, and returns the index of the new entry to the application. Similarly for *close* on a file descriptor, the emulator deallocates the corresponding send right. If this is the last reference to the file port, then a *no_senders_notification* is generated for this port, and the concerned file server will clean up the corresponding file table entry. However, *dup* and *close* on the socket descriptors result in messages to RLS.

4.8.3 Sync

There are some system calls related to the file services, for which messages should be sent to all the file servers in the system. For example, the *sync* system call forces the dirty (modified) buffers in the buffer cache to be written to the disk. Since this

call applies to all the mounted file systems, it should be sent to all the file servers. In order to do this, emulator sends the sync message to RLS, which forwards it to the contact ports of all the file servers currently running. As mentioned earlier, RLS obtains this information from *FS_Table*.

Chapter 5

Performance

We have conducted some experiments in order to measure the performance of our implementation relative to that of NetBSD and LITES. In this chapter, we present the results of these experiments and analyze the results.

5.1 Experimental Setup

We used a 486 DX2 66 MHz processor with 8MB of main memory to run the benchmarks. All the measurements were taken on a file system with 1K block size. We obtained the time intervals by using the *gettimeofday* system call before and after the call whose performance is being measured. All the times reported are in the units of microseconds (μs). The hard disk had the following characteristics.

Number of bytes/sector = 512
Number of sectors/track = 40
Number of tracks/cylinder = 14
Number of cylinders = 944
Disk head RPM = 3600

We took the measurements for three different architectures on the same machine by booting it with each system in turn. They are NetBSD-1.0, Lites-1.1.u2, and our system (LitesNew). We expect NetBSD to perform quite well compared to the other two, as it is a monolithic architecture. But the performance of our system

should be reasonable compared to that of Lites since both are similar in that they are emulations on Mach.

5.2 Results and their Analysis

The system calls whose performance is measured fall broadly into two categories. They are calls that contain pathname such as *open* and those that work on file descriptors such as *read*.

5.2.1 Calls containing Pathname

In measuring the performance of the calls containing a pathname, we considered three cases. They are as follows.

- Fifty different files were opened and the average time was calculated after ignoring the maximum and minimum values. This was repeated 5 times and the average was taken. The file names did not span different file systems. The system was booted each time before taking these measurements in order to negate the effect of the buffer cache. Since each file was opened for the first time, their inodes would have to be freshly read from the disk. Thus the values we get in this experiment indicate the time it takes to open a file whose inode is not present in the buffer cache.

- A file is opened 50 times and the average is calculated after ignoring the maximum and minimum values. We repeat the same procedure 5 times with booting the system each time. The file name does not span different file systems. The initial open costs relatively more compared to further open calls because initially the corresponding inode has to be read from the disk, while for the next cases, it will be found in the buffer cache. By ignoring the maximum value, we are ignoring the cost of this initial open. Thus the value we get here is the average time for opening a file whose inode is already present in the buffer cache.

- The same pattern as above is followed except that the file name crosses a mount point. As already explained, in our system, this results in forwarding of the system call message.

We took measurements for the system calls *open* and *stat*. These results are shown in Table 1 and Table 2 respectively.

	NetBSD	Lites	LitesNew
Same file, without crossing mount points	221 (2)	613 (29)	3189 (135)
Same file, crossing mount points	220 (2)	689 (33)	3648 (147)
Different files	732 (3)	1939 (99)	4260 (180)

Table 1: Comparison of times (μs) for *open* (numbers shown in parenthesis are the standard deviations of measurements)

	NetBSD	Lites	LitesNew
Same file, without crossing mount points	181 (2)	572 (20)	1510 (67)
Same file, crossing mount points	181 (2)	735 (31)	1531 (72)
Different files	678 (2)	1714 (78)	2347 (119)

Table 2: Comparison of times (μs) for *stat* (numbers shown in parenthesis are the standard deviations of measurements)

As was expected, NetBSD performs quite well with respect to Lites and LitesNew. Comparison between the figures in the last two columns shows that our system took lot more time than Lites in each case. This is due to the involvement of an extra message in our architecture. In the calls *open* and *stat*, the file servers need to have the process credentials, for which, they send a message to RLS. The time taken for this RPC adds to the cost of these calls, thereby increasing their overall time. Moreover, in the case of file names crossing mount points, the system call message is forwarded.

5.2.2 Calls on File descriptors

- *Read*

we measured the performance of the *read* system call for three different sizes of reads — 1K, 4K, and 8K. A file is read sequentially, and when the end of the file is reached, another file is opened and read. We continued this process till 100 reads are completed, and then, the average is taken after ignoring the maximum and minimum values. This procedure is repeated 5 times, booting the system each time, and the average value is determined. The results are shown in Table 3.

Size	NetBSD	Lites	LitesNew
1K	1325 (37)	3674 (131)	3184 (116)
4K	2921 (55)	12612 (410)	8551 (287)
8K	5365 (74)	16490 (593)	14837 (372)

Table 3: Comparison of times (μs) for *read* (numbers shown in parenthesis are the standard deviations of measurements)

NetBSD performs far better than either Lites or our system, because of the reasons already explained. For a *read* system call, the emulator in our system sends a message to the corresponding file port. Thus both Lites and LitesNew need only one message for a *read*, and therefore, their performances should not differ much. This is true, as can be seen, from the figures in the last two columns. In fact, our system performed consistently better than Lites for all the three different sizes.

- *Write*

Usually, the *write* system call only puts the data in the system buffer before returning to the user process. The dirty buffers in the buffer cache are periodically written to the disk. These are known as asynchronous writes. But some applications need to be sure that the data is stored on the disk, before proceeding. There is an option to the *write* system call which causes the kernel to write to the disk before returning to the application. These writes are called synchronous writes. We measured the

times for both kinds of writes.

The performance of the *write* system call is taken for three different sizes — 1K, 4K, and 8K. We created 10 files, writing in each file 10 times and thus measuring times for 100 writes. Their average is taken after ignoring maximum and minimum values. Table 4 shows these results for asynchronous writes while Table 5 details the results for synchronous writes.

Size	NetBSD	Lites	LitesNew
1K	342 (5)	1102 (56)	1408 (65)
4K	731 (4)	1816 (83)	2122 (84)
8K	776 (7)	11306 (476)	8551 (230)

Table 4: Comparison of times (μs) for *asynchronous write* (numbers shown in parenthesis are the standard deviations of measurements)

Size	NetBSD	Lites	LitesNew
1K	344 (4)	959 (45)	1191 (54)
4K	733 (4)	1714 (43)	1863 (78)
8K	771 (11)	10265 (478)	7155 (344)

Table 5: Comparison of times (μs) for *synchronous write* (numbers shown in parenthesis are the standard deviations of measurements)

As in the case of *read*, the *write* system call requires only one message in both Lites and LitesNew, and therefore, their performances should not differ much. This is indeed the case with our system taking slightly more time for writes of sizes 1K and 4K. But in the case of writes of size 8K, our system did better compared to Lites for both asynchronous and synchronous writes.

5.3 Conclusions

As can be observed, NetBSD performs better in all the cases for all the system calls considered. This is not surprising since no RPCs are involved in the system call

invocation in NetBSD, while in the other two systems, a message has to be sent for this purpose.

The system calls in both Lites and LitesNew are implemented by sending a message to a server. In Lites, the emulator sends all the messages to the Lites server, whereas in our system, file system related calls are sent to the file servers. As far as the calls on the file descriptors are concerned, as already explained, only one message needs to be sent in both Lites and LitesNew. Thus we expected their performances to be similar, which is the case with the measurements obtained. In fact, for the reads of all sizes, and writes of the size 8K, the performance of our system was better than that of Lites. This may be due to the reduction in some overheads in the system because of isolating the file services in a separate server.

Regarding the case of calls containing pathname, extra messages result in our system. This is due to the requirements of process credentials and the need for forwarding the message while crossing different file systems. Thus our system, as shown in the measurements above, took lot more time compared to Lites for these calls. But, since *read* and *write* constitute majority of file operations, and as they need only one message, we expect the overall performance of our system to be reasonable with respect to Lites.

Chapter 6

Conclusions

In this thesis, we presented a design for Unix emulation over Mach where the file services are isolated into a separate component, and an implementation of the file server for FFS.

Traditional Unix is a monolithic kernel which provides all the services such as files, sockets, and processes. Due to its monolithic structure, it is very difficult to add new services to it. Moreover, debugging and testing of new services requires disrupting the existing system. In an effort to improve the modularity of the system, the idea of grouping together the most basic abstractions in a *microkernel* and running all the higher level services such as files and sockets in user-level servers, gained ground. Mach is one such microkernel and Lites is a 4.4 BSD compatible single server emulation over Mach. In Lites, an emulator is mapped in each application's address space at a fixed location. The system calls invoked by the application trap into the Mach kernel, which redirects the call to an address in the user process. The emulator lies at this address, decodes the call, communicates with the server if required, and replies to the application. Thus the applications find a seamless Unix environment though it is being provided by a server and an emulator.

Even though this system is flexible in that different servers can be run simultaneously to provide different operating system environments, the problem of lack of modularity in the Lites server remains. This is because the Lites design is still monolithic by keeping all the Unix functionality in one server. We wanted to have a

modular Unix emulation over Mach by having different servers provide logically separate functionalities of the Unix system. The presented work is a beginning of our efforts in this direction. Instead of starting from scratch and design the complete multi-server emulation, we have chosen to start with an existing Unix emulation server (Lites) and isolate different functionalities one by one.

In this thesis, we have presented the design for a server which provides file services, and together with the reduced Lites server, provides Unix system functionality to the applications. In our system, there is a separate file server for each type of file system, as we did not want to have a single "universal" file server providing the functionality of all the file system types. This would have defeated the very purpose of our project of having a modular Unix emulation. We also presented the details of an implementation of a file server for FFS (Fast File System). All the file servers cooperate among themselves to provide the illusion of single file name space to the applications. The file servers for all the file system types supported by the system need not be run at boot time. It is enough to start a file server to manage the root file system. A file server is automatically started when a mount request for the corresponding type of file system is received. In our architecture, new file system types can be added without disrupting the existing system. This is achieved by keeping the file server image in a standard place, so that it can be started when the corresponding type of file system is mounted. Thus the system is modular in that it can be easily enhanced by adding new file system types without disrupting the existing system.

The file servers contact RLS for obtaining process context information such as process credentials. The system calls which contain a pathname need extra messages in our system than in Lites. This is because of the requirements of process credentials and the need for forwarding the message while crossing different file systems. Thus our system performs poorly for these calls with respect to Lites. But those calls that operate on file descriptors require only one message in both the systems, and therefore, their performances should not differ much. We presented the performance measurements of our system with respect to that of Lites, which validated this hypothesis. Since *read* and *write* constitute majority of file operations, and as they

need only one message, we expect the overall performance of our system to be reasonable with respect to Lites.

6.1 Future Work

Since in our system, a file server provides the functionality of a single type of file system, there is no need for the VFS layer code. But, in the present implementation of the FFS server, the VFS layer code is not removed. Its removal will reduce the overhead further and may improve the system performance. Similarly, in the present implementation, code for supporting process limits and Unix domain sockets is missing.

As mentioned before, the presented work is a beginning of our efforts in building the multi-server Unix emulation over Mach. The ultimate goal is to have a true multi-server implementation where each server performs a specific specialized job. The next step in this process will be to isolate some other functionality from the Lites server such as socket services. File servers for different types of file systems also need to be implemented.

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–112, Atlanta, GA, Summer 1986. USENIX.

[B+93] Boykin et al. *Programming under Mach.* System Programmer's series. Addison-Wesley, Reading, MA, first edition, 1993.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1986.

[G+90] Golub et al. Unix as an application program. In *Proceedings of the Summer Usenix Conference*, pages 87–96, 1990.

[Hel94] Johannes V. Helander. Unix under Mach, The LITES Server. Master's thesis, Helsinki University of Technology, Helsinki, 1994.

[Hil92] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 27-28 1992. USENIX.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, USA, 1989.

[Loe91] Keith Loepere. *Server Writer's Guide.* OSF and CMU, 1991.

CENTRAL LIBRARY
I. I. T., KANPUR
No. A 124940

[Roz92] Marc Rozier. Chorus (overview of the chorus distributed operating system). In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, April 27-28 1992. USENIX.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings*, pages 119–130, Portland, OR, Summer 1985. USENIX.

[Sta90] Richard Stallman. Towards a New Strategy in OS Design. In the GNU Bulletin, http://www.gnu.org, 1990.

[vR92] Robbert van Renesse. Amoeba. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 1–10, Seattle, WA, April 27-28 1992. USENIX.